Cvičení 2: Rekurze a seznamy

Před druhým cvičením je zapotřebí znát:

- zápis funkce definované pomocí vzorů;
- zápis seznamů pomocí výčtu prvků, tj. například `[1, 2, 10]`;
- základní funkce pro práci se seznamy, tj. například

```
(:) :: a -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
head :: [a] -> a
tail :: [a] -> [a]
```

- základní vzory pro seznamy, tj. například `[]`, `(x:xs)`, `[x]`, `[x, y]`.

Etudy

Etuda 2.η.1 Určete typy následujících seznamů. Nejprve intuitivně, následně je ověřte v interpretu.

a) `["a", "b", "c"]`
b) `['a', 'b', 'c']`
c) `"abc"`
d) `'a' : 'b' : 'c' : []`
e) `[(True, ()), (False, ())]`
f) `[(++) "a" "b", "X" ++ "Y"]`
g) `[(&&), (||)]`
h) `[]`
i) `[[]]`
j) `[[], [True]]`

Etuda 2.η.2 Bez použití jakýchkoli knihovních funkcí definujte funkci `isEmpty` typu `[a] -> Bool`, která pro prázdný vstupní seznam vrátí `True` a pro neprázdný vrátí `False`.

Etuda 2.η.3 Bez použití knihovních funkcí napište funkci `myHead` typu `[a] -> a`, která vrátí první prvek zadaného neprázdného seznamu, a funkci `myTail` typu `[a] -> [a]`, která pro zadaný neprázdný seznam vrátí tentýž seznam bez prvního prvku.

Etuda 2.η.4 Napište funkci `neck :: a -> [a] -> a`, která vrátí druhý prvek zadaného seznamu, případně „náhradní“ prvek, je-li zadaný seznam prázdný nebo jednoprvkový. Nevyužívejte knihovní funkce, pouze definice podle vzoru.

Etuda 2.η.5 Pro následující vzory a seznamy určete, které vzory mohou reprezentovat které seznamy. Stanovte, jak se navážou proměnné ze vzoru.

- Vzory: `[]`, `x`, `[x]`, `[x, y]`, `(x : s)`, `(x : y : s)`, `[x : s]`, `((x : y) : s)`
- Seznamy: `[1]`, `[1, 2]`, `[1, 2, 3]`, `[[]]`, `[[1]]`, `[[1], [2, 3]]`

2.1 Rekurze na číslech

Př. 2.1.1 Bez použití funkcí `mod`, `div`, `even` a `odd` naprogramujte rekurzivní funkci `isEven`, která pro dané nezáporné celé číslo rozhodne, jestli je sudé. Určete i její typ. Záporná čísla neřešte. Nejprve se zamyslete: jak se dá určit sudost čísla na základě sudosti menšího čísla?

Protože zadání omezuje vstupy na nezáporná čísla, nemusíte se chováním funkce na záporných vstupech trápit. Pro zajímavost si ale zkuste funkci se záporným argumentem vyhodnotit a chování vysvětlit. Pro násilné zastavení nekončícího výpočtu použijte klávesovou zkratku Ctrl+C (mnemotechnická pomůcka: *cancel*). Vyzkoušejte si i rozdíl mezi `isEven -1` a `isEven (-1)` a vysvětlete jej.

Př. 2.1.2 Bez použití knihovní funkce `mod` naprogramujte rekurzivně funkci `mod3`, která pro dané nezáporné celé číslo vypočítá jeho zbytek po dělení 3. Nezapomeňte uvést i její typ. Například:

```
mod3 0 ⇝* 0        mod3 7 ⇝* 1
mod3 5 ⇝* 2        mod3 9 ⇝* 0
```

*
* *

Typové třídy. Minule jsme viděli, že v typových signaturách funkcí pracujících s čísly se nemusíme omezovat jen konkrétní typy (jako `Integer`), ale můžeme využívat *typové třídy*, které nám dovolí do funkcí předávat hodnoty různých, ale v něčem podobných typů. Vzpomeňte si na typovou třídu `Num` pro libovolná čísla a `Integral` pro celočíselné typy. Například předchozí funkce by mohla mít obecnější typ

```
mod3 :: Integral i => i -> i
```

nebo dokonce

```
mod3 :: (Integral i1, Integral i2) => i1 -> i2
```

V následujících příkladech již na vhodných místech místo konkrétních (tzv. *monomorfních*) číselných typů používejte *polymorfní* typy omezené typovými třídami. Jeden z obvyklých případů, kde se přesto používá monomorfní `Int`, jsou různé indexy, počty a délky.

Pan Fešák doporučuje: Informace o typových třídách a v nich definovaných funkcích můžeme zjistit pomocí příkazu `:i` v GHCi. Např.

```
> :i Fractional
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
        -- Defined in ‘GHC.Real’
instance Fractional Float -- Defined in ‘GHC.Float’
instance Fractional Double -- Defined in ‘GHC.Float’
```

To nám říká, že typová třída `Fractional` obsahuje funkce `(/)`, `recip` a `fromRational`, že do ní patří typy `Float` a `Double` a že je-li něco ve třídě `Fractional`, pak je to nutně také ve třídě `Num`. Krom toho nám tento výpis také dává informace o tom, které funkce bychom minimálně museli implementovat, pokud bychom chtěli do této třídy nějaký nový typ přidat.

Chceme-li zjistit, zda jde nějaký kontext, například `(Num a, Floating a)` zjednodušit, potřebujeme zjistit, zda některá z těchto typových tříd vyžaduje některou další. V tomto případě bude potřeba navíc jít přes mezikrok:

```
class Fractional a => Floating a where {- ... -}
class Num a => Fractional a where {- ... -}
```

Tedy pokud je nějaký typ ve `Floating`, musí být nutně ve `Fractional` a tedy i v `Num`. Kontext tedy zjednodušíme na `Floating a`.

Př. 2.1.3 2.1.div »= Bez použití knihovní funkce `div` naprogramujte rekurzivně funkci `div3`, která dané nezáporné celé číslo celočíselně vydělí třemi. Nezapomeňte uvést typ s vhodnou typovou třídou. Například:

```
div3 0 ⇝* 0
div3 5 ⇝* 1
div3 7 ⇝* 2
div3 9 ⇝* 3
```

Př. 2.1.4 Implementujte rekurzivní funkci pro výpočet faktoriálu a určete její typ.

Př. 2.1.5 Definujte funkci `power` takovou, že `power x n` se pro nezáporné celé číslo `n` vyhodnotí na `x` na `n`-tou. Všimněte si, že `x` nemusí být celé číslo. Začněte napsáním typové signatury s vhodnými typovými třídami. Příklad vyhodnocení:

```
power 3 1 ⇝* 3
power 3 2 ⇝* 9
power 3 3 ⇝* 27
power 3 0 ⇝* 1
power 0 3 ⇝* 0
power 0.5 3 ⇝* 0.125
```

*
* *

Predikát je funkce vracející `Bool` podle vlastností svých argumentů.

Př. 2.1.6 »= Napište predikát `isPower2`, který o zadaném přirozeném čísle rozhodne, jestli je mocninou dvojky. Nezapomeňte funkci otypovat s využitím vhodné typové třídy. Mohou se vám hodit funkce `even` a `odd`. Například:

```
isPower2 0 ⇝* False
isPower2 1 ⇝* True
isPower2 2 ⇝* True
isPower2 6 ⇝* False
isPower2 8 ⇝* True
isPower2 9 ⇝* False
```

Nápověda: Mocnina dvojky je dvojnásobkem jiné mocniny dvojky.

Př. 2.1.7 Definujte funkci `digitsSum`, která po aplikaci na kladné celé číslo vrátí jeho ciferný součet. Nezapomeňte na typ. Můžete použít funkce `div` a `mod`. Například:

```
digitsSum 123 ⇝* 6
digitsSum 103 ⇝* 4
```

Př. 2.1.8 Napište funkci `mygcd`, která po aplikaci na dvě kladná celá čísla vrátí jejich největšího společného dělitele. Nepoužívejte funkci `gcd`. Pokuste se o co nejefektivnější implementaci.

```
mygcd 18 10 ⇝* 2
mygcd  9 27 ⇝* 9
mygcd  4 16 ⇝* 4
mygcd 15  8 ⇝* 1
```

Př. 2.1.9 Naprogramujte funkci `primeDivisors`, která rozhodne, součinem kolika prvočísel je zadané kladné číslo. Můžete použít funkce `div` a `mod`. Například:

```
primeDivisors 3  ⇝* 1
primeDivisors 4  ⇝* 2
primeDivisors 5  ⇝* 1
primeDivisors 6  ⇝* 2
primeDivisors 8  ⇝* 3
primeDivisors 12 ⇝* 3
primeDivisors 15 ⇝* 2
primeDivisors 1  ⇝* 0
```

Př. 2.1.10 Definujte funkce `plus` a `times`, které budou ekvivalentní operátorům `(+)` a `(*)` na přirozených číslech. Nepoužívejte vestavěné `(+)` ani `(*)`. Můžete však používat libovolné jiné funkce, doporučujeme podívat se zejména na funkce `pred` a `succ` (jejich typ je ve skutečnosti o něco obecnější, ale můžete uvažovat, že to je `Integral i => i -> i`).

Bonus: implementujte funkce `plus'` a `times'`, které budou fungovat na všech celých číslech.

Př. 2.1.11 Co počítá následující funkce? Jak se chová na argumentech, kterými jsou nezáporná čísla? Jak se chová na záporných argumentech?

```
fun 0 = 0
fun n = fun (n - 1) + 2 * n - 1
```

2.2 Rekurze na seznamech

Př. 2.2.1 Napište nerekurzivní funkci, která na začátek zadaného seznamu čísel vloží hodnotu 42. Jaký má vaše funkce typ? Nezapomeňte na existenci typových tříd!

Př. 2.2.2 Bez použití knihovní funkce `last` definujte funkci `getLast :: [a] -> a`, která vrátí poslední prvek neprázdného seznamu.

Př. 2.2.3 Bez použití funkce `init` definujte funkci `stripLast` typu `[a] -> [a]`, která pro neprázdný seznam vrátí tentýž seznam bez posledního prvku.

Př. 2.2.4 Bez použití knihovní funkce `length` definujte funkci `len :: [a] -> Integer`, která spočítá délku zadaného seznamu.

Př. 2.2.5 2.2.nth Napište funkci `nth :: Int -> [a] -> a`, která ze zadaného seznamu vrátí prvek na pozici určené prvním argumentem funkce. Počítá se od nuly. Můžete předpokládat, že vstupní seznam má dostatečnou délku. Nepoužívejte knihovní funkce. Například:

```
nth 0 [4, 3, 5] ⇝* 4
nth 1 [4, 3, 5] ⇝* 3
nth 2 [4, 3, 5] ⇝* 5
nth 2 "Get Schwifty" ⇝*
↪ 't'
```

*
**

> Typová třída `Eq` sdružuje typy, jejichž hodnoty můžeme porovnávat operátory `(==)` a `(/=)`. Tuto typovou třídu potřebujeme i pro využití vzorů s konkrétními hodnotami čísel (typová třída `Num` negarantuje `Eq`, `Integral` však již ano).

Př. 2.2.6 Napište funkci `contains :: Eq a => [a] -> a -> Bool`, která vrací `True`, pokud seznam v prvním argumentu obsahuje prvek zadaný druhým argumentem, jinak vrací `False`. Nepoužívejte funkci `elem` a jí podobné. Například:

```
contains [1, 2, 3, 4] 42 ⇝* False
[1, 2, 3] `contains`  2 ⇝* True
[]        `contains`  () ⇝* False
"TEAM"    `contains` 'I' ⇝* False
```

Př. 2.2.7 Napište funkci `containsNtimes` typu `Eq a => Integer -> a -> [a] -> Bool` takovou, že `containsNtimes n x xs` bude `True` právě tehdy, když seznam `xs` obsahuje alespoň `n`

výskytů hodnoty `x`. Například:

```
containsNtimes 2 42 [1, 2, 42] ⇝* False
containsNtimes 2 42 [1, 42, 2, 42] ⇝* True
containsNtimes 3 42 [1, 42, 2, 42] ⇝* False
containsNtimes 0 42 [1, 2] ⇝* True
containsNtimes 2 'l' "Haskell" ⇝* True
```

Př. 2.2.8 Mějme neprázdný seznam typu `[(String, Integer)]`, který reprezentuje seznam jmen studentů s jejich počty bodů z předmětu IB015. Naprogramujte

a) funkci `getPoints :: String -> [(String, Integer)] -> Integer`, která vrátí počet bodů studenta se jménem zadaným v prvním argumentu (nebo 0, pokud takový student v seznamu není),

b) funkci `getBest :: [(String, Integer)] -> String`, která vrátí jméno studenta s nejvíce body.

Například tedy:

```
getPoints "Stan" [("Kyle", 30), ("Eric", 42), ("Stan", 20)] ⇝*
↪ 20
getPoints "Tomas" [("Kyle", 30), ("Eric", 42), ("Stan", 20)] ⇝*
↪ 0
getBest [("Kyle", 30), ("Eric", 42), ("Stan", 20)] ⇝* "Eric"
```

Př. 2.2.9 2.2.app Napište funkci `append :: [a] -> [a] -> [a]`, jejíž výsledek pro dva seznamy bude seznam, který vznikne zřetězením těchto dvou seznamů. Nepoužívejte knihovní operátor `(++)`. Například:

```
append [1, 2] [3, 4, 8] ⇝* [1, 2, 3, 4, 8]
append [] [3, 4] ⇝* [3, 4]
append [3, 4] [] ⇝* [3, 4]
append "Legen" "dary" ⇝* "Legendary"
```

Nápověda: `[1,2] ++ [3,4] = [1,2,3,4] = 1:2:3:4:[] = 1:2:[3,4]`

Př. 2.2.10 Napište funkci `pairs :: [a] -> [(a, a)]`, která bere prvky ze vstupního seznamu po dvou prvcích a vytváří seznam dvojic těchto prvků. Pokud má seznam lichý počet prvků, poslední prvek se zahodí. Například:

```
pairs [4, 8, 15, 16, 23] ⇝* [(4, 8), (15, 16)]
pairs [4, 8, 15, 16, 23, 42] ⇝* [(4, 8), (15, 16), (23, 42)]
pairs "Humphrey" ⇝* [('H', 'u'), ('m', 'p'), ('h', 'r'), ('e',
↪ 'y')]
```

Př. 2.2.11 Napište následující funkce pracující se seznamy čísel pomocí rekurze a vzorů:

a) `listSum :: Num n => [n] -> n`, která dostane seznam čísel a vrátí součet všech jeho prvků.

b) `oddLength :: [a] -> Bool`, která vrátí `True`, pokud je seznam liché délky, jinak `False` (bez použití funkce `length`).

c) `add1 :: Num n => [n] -> [n]`, která každé číslo ve vstupním seznamu zvýší o 1,

d) `multiplyN :: Num n => n -> [n] -> [n]`, která každé číslo ve vstupním seznamu vynásobí prvním argumentem funkce,

e) `deleteEven :: Integral i => [i] -> [i]`, která ze seznamu čísel odstraní všechna sudá čísla,

f) `deleteElem :: Eq a => a -> [a] -> [a]`, která ze seznamu odstraní všechny výskyty hodnoty zadaného prvním argumentem,

g) `largestNumber :: [Integer] -> Integer`, vrátí největší číslo ze zadaného neprázdného seznamu čísel,
h) `listsEqual :: Eq a => [a] -> [a] -> Bool`, která dostane na vstup dva seznamy a vrátí `True` právě tehdy, když se rovnají (bez použití funkce `(==)` na seznamy),
i) `multiplyEven :: [Integer] -> [Integer]`, která vezme seznam čísel a vrátí seznam, který bude obsahovat všechna sudá čísla původního seznamu vynásobená 2 (lichá čísla vynechá),
j) `sqroots :: [Double] -> [Double]`, která ze zadaného seznamu vybere kladná čísla a ta odmocní (může se vám hodit funkce `sqrt`).

Př. 2.2.12 2.2.every 
Napište funkci `everyNth :: Integer -> [a] -> [a]` takovou, že seznam `everyNth n xs` bude obsahovat každý n-tý prvek ze seznamu `xs`. Například:

```
everyNth 2 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 1, 2, 7]
everyNth 3 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 3, 7]
everyNth 4 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 2]
everyNth 1 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 8, 1, 3, 2, 5, 7]
everyNth 2 "BoJack Horseman" ⇝* "BJc osmn"
```

Př. 2.2.13 
Napište funkci `brackets :: String -> Bool`, která dostane řetězec složený ze znaků `'('` a `')'` a rozhodne, jestli se jedná o korektní uzávorkování. Například:

```
brackets "(()())" ⇝* True
brackets "(()()" ⇝* False
brackets "())()" ⇝* False
brackets "())(" ⇝* False
brackets "" ⇝* True
```

Př. 2.2.14
Zadefinujte predikát `palindrome`, který o řetězci na vstupu rozhodne, jestli je palindrom. Napište také funkci `palindromize`, která ze zadaného řetězce udělá palindrom tak, že na jeho konec doplní co nejméně znaků. Tedy například:

```
palindrome "lol" ⇝* True
palindrome "ABBA" ⇝* True
palindrome "brienne" ⇝* False
palindromize "brienne" ⇝* "brienneirb"
```

Př. 2.2.15 
Napište funkci `getMiddle :: [a] -> a`, která pro zadaný neprázdný seznam vrátí jeho prostřední prvek bez zjišťování jeho délky. Pokud má seznam sudý počet prvků, vraťte levý z prostředních dvou. Například:

```
getMiddle [1] ⇝* 1
getMiddle [2, 1] ⇝* 2
getMiddle [2, 1, 5] ⇝* 1
getMiddle [2, 1, 5, 6] ⇝*
 ↪ 1
getMiddle [2, 1, 5, 6, 3]
 ↪ ⇝* 5
getMiddle "Don't blink!"
 ↪ ⇝* ' '
```

Nápověda: Pokud zajíc běží dvakrát rychleji než želva, pak v okamžiku, kdy zajíc vyhrál závod, je želva v polovině trati.

*
* *

Na konci druhého cvičení byste měli umět:

- definovat vlastní rekurzivní funkce pracující s celými čísly;

- pracovat se seznamy a definovat na nich funkce pomocí vzorů;
- definovat rekurzivní funkce na seznamech;
- použít v typech svých funkcí základní typové třídy jako `Num` a `Eq`.

Řešení

Řeš. 2.η.1 Použijte příkaz `:t` k otypování výrazu v `ghci` (typ `[Char]` je ekvivalentní typu `String`).

a) `[[Char]]` (což je stejné jako `[String]`)
b) `[Char]` (což je stejné jako `String`)
c) `[Char]` (což je stejné jako `String`)
d) `[Char]` (což je stejné jako `String`)
e) `[(Bool, ())]`
f) `[String]`, při otypování takovýchto výrazů je třeba si dát pozor. Výraz sice obsahuje funkci `++`, která má v tomto kontextu typ `String -> String -> String`, avšak `String -> String -> String` není výsledný typ, protože funkci už byly dodány argumenty, a tedy typ prvků v seznamu je `String`.
g) `[Bool -> Bool -> Bool]`
h) `[a]`, z výrazu nevyplývá žádné omezení na typ prvků, který může obsahovat, proto je typ prvků úplně obecný, tedy `a`.
i) `[[a]]`, podobně jako v předešlém případě, žádné omezení na typ prvků vnitřního seznamu.
j) `[[Bool]]` typové omezení vzniká kvůli konkrétní hodnotě ve druhém prvku.

Řeš. 2.η.2 Stačí použít definici funkce podle vzorů. Prázdný seznam je prázdný, žádný jiný seznam není prázdný (duh).

```
isEmpty :: [a] -> Bool
isEmpty [] = True
isEmpty _  = False
```

Řeš. 2.η.3 Opět stačí použít definici funkce podle vzorů. Případy, kdy vstupem funkce je prázdný seznam, buď není potřeba vůbec definovat, nebo lze použít funkci `error` nebo hodnotu `undefined`.

```
myHead :: [a] -> a
myHead (x : _) = x
myHead []      = error "myHead: Empty list."

myTail :: [a] -> [a]
myTail (_ : xs) = xs
myTail []       = error "myTail: Empty list."
```

Řeš. 2.η.5

- `[]`
  Tento vzor představuje prázdný seznam. Nemůže reprezentovat žádný z uvedených seznamů.
- `x`
  Na tento vzor se může navázat libovolná hodnota, a tedy zejména libovolný ze zadaných seznamů.
- `[x]`
  Představuje libovolný jednoprvkový seznam. Z uvedených může reprezentovat seznamy `[1]`, `[[]]`, `[[1]]`.
- `[x, y]`
  Představuje libovolný dvouprvkový seznam. Z uvedených může reprezentovat seznamy `[1, 2]`, `[[1], [2, 3]]`.
- `(x : s)`

Libovolný neprázdný seznam. Proměnná `x` reprezentuje první prvek, proměnná `s` seznam ostatních prvků. Tento vzor může reprezentovat všechny uvedené seznamy (ano, i `[[]]`).

- `(x : y : s)`
  Představuje libovolný seznam, který má alespoň 2 prvky. Proměnná `x` reprezentuje první prvek, `y` druhý prvek a `s` seznam ostatních prvků. Z uvedených může reprezentovat seznamy `[1, 2]`, `[1, 2, 3]`, `[[1], [2, 3]]`.
- `[x : s]`
  Jednoprvkový seznam, jehož jediným prvkem je neprázdný seznam. Proměnná `x` reprezentuje první prvek vnitřního seznamu, proměnná `s` seznam ostatních prvků vnitřního seznamu. Z uvedených může reprezentovat pouze seznam `[[1]]`.
- `((x : y) : s)`
  Představuje neprázdný seznam, jehož prvním prvkem je neprázdný seznam. Proměnné `x` a `y` reprezentují první prvek prvního prvku a seznam ostatních prvků prvního prvku, proměnná `s` reprezentuje ostatní prvky vnějšího seznamu. Z uvedených může reprezentovat seznamy `[[1]]`, `[[1], [2, 3]]`.

Řeš. 2.1.1 Myšlenka řešení může být následující: 0 je sudá, 1 není sudá, a každé jiné kladné číslo je sudé právě tehdy, když číslo o 2 menší je sudé.

```
isEven :: Integer -> Bool
isEven 0 = True
isEven 1 = False
isEven x = isEven (x - 2)
```

Vymyslet se dá i řešení snižující parametr o jedna a používající funkci `not`.

Výraz `isEven -1` skončí s typovou chybou, neboť minus se přednostně chápe jako binární operátor a jedná se tak o odečtení jedničky od funkce. V případě `isEven (-1)` se již jedná skutečně o zápornou jedničku a při definici výše dojde k zacyklení, neboť se funkce bude vždy rekurzivně volat s menším argumentem (podle posledního řádku) a nikdy se nezastaví na některém z prvních dvou vzorů.

Řeš. 2.1.2 Myšlenka řešení může být následující: zbytek 0 po dělení třemi je 0, zbytek 1 po dělení třemi je 1, zbytek 2 po dělení třemi je 2 a zbytek dělení třemi pro každé jiné kladné číslo je stejný, jako zbytek po dělení třemi pro číslo o 3 menší.

```
mod3 :: Integer -> Integer
mod3 0 = 0
mod3 1 = 1
mod3 2 = 2
mod3 x = mod3 (x - 3)
```

Řeš. 2.1.3 Myšlenka je podobná jako v předešlém příkladu. Jen se zde počítá, kolikrát je potřeba odečíst 3, aby se argument dostal do intervalu $\langle 0, 3\rangle$.

```
div3 :: Integral i => i -> i
div3 0 = 0
div3 1 = 0
div3 2 = 0
div3 x = 1 + div3 (x - 3)
```

Řeš. 2.1.4

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n - 1)
```

Funkce je opět definována po částech. Předpokládáme, že dostane jako argument jenom

nezáporné celé číslo. Pokud je argument 0, výsledek je zřejmě 1. Pokud je naopak argument kladné číslo, víme, že $n! = n \times (n-1)!$, kde druhý výsledek získáme pomocí rekurzivního volání. Poznamenejme, že závorky kolem `n - 1` je nutno použít, protože jinak by se výraz implicitně uzávorkoval jako `(fact n) - 1`, protože aplikace prefixově zapsané funkce má vyšší prioritu než infixové operátory.

Řeš. 2.1.5 Bázový případ $z^0 = 1$. Pro každé kladné n naopak platí $z^n = z \cdot z^{n-1}$, přičemž hodnota z^{n-1} jde vypočítat rekurzivně.

```
power :: (Num n, Integral i) => n -> i -> n
power _ 0 = 1
power x n = x * power x (n - 1)
```

Všimněte si, že definovaná funkce vždy použije n rekurzivních volání. Tento počet se ale dá zmenšit, když si uvědomíme, že při každém rekurzivním volání není potřeba n snižovat jen o 1, ale je možné ho zmenšit přibližně na polovinu. Platí totiž

- $z^0 = 1$
- $z^{2n} = (z^n)^2$
- $z^{2n+1} = (z^n)^2 \cdot z$

Následující implementace, která tuto myšlenku využívá, tedy potřebuje jen přibližně $\log(n)$ rekurzivních volání.

```
power' _ 0 = 1
power' z n = if even n then half * half else half * half * z
    where half = power' z (n `div` 2)
```

Řeš. 2.1.6 Stačí si uvědomit, že číslo je mocninou 2 právě tehdy, když je 1 nebo je sudé a jeho polovina je mocninou 2.

```
isPower2 :: Integral i => i -> Bool
isPower2 0 = False
isPower2 1 = True
isPower2 x = even x && isPower2 (div x 2)
```

Řeš. 2.1.7

```
digitsSum :: Integral i => i -> i
digitsSum 0 = 0
digitsSum x = x `mod` 10 + digitsSum (x `div` 10)
```

Řeš. 2.1.8 K řešení můžeme použít známý Euklidův algoritmus. Konkrétně použijeme jeho rekurzivní verzi, která využívá zbytky po dělení.

```
mygcd :: Integral i => i -> i -> i
mygcd x 0 = x
mygcd x y = mygcd (min x y) ((max x y) `mod` (min x y))
```

Řeš. 2.1.9 Příklad lze vyřešit i bez generování všech prvočísel. Stačí zadané číslo dělit 2 tolikrát, kolikrát je to možné, a zároveň přičítat za každé vydělení k výsledku jedničku. Poté číslo budeme dělit 3, 4, 5, 6, atd., dokud po dělení nezbyde výsledek 1. Ačkoliv například čísla 4 a 6 nejsou prvočísla, ale přesto jimi dělíme, není to problém, protože pokud jsme číslo už vydělili 2 a 3, kolikrát to bylo možné, nemůže už být dělitelné ani 4 ani 6.

```
primeDivisors :: Integral i => i -> Integer
primeDivisors n = divisorsFrom n 2
```

```
    where
        divisorsFrom 1 _ = 0
        divisorsFrom n d = if mod n d == 0
                           then 1 + divisorsFrom (n `div` d) d
                           else divisorsFrom n (d + 1)
```

Řeš. 2.1.10

```
plus :: Integral i => i -> i -> i
plus 0 y = y
plus x y = plus (pred x) (succ y)

times :: Integral i => i -> i -> i
times 0 y = 0
times x 0 = 0
times 1 y = y
times x y = plus y (times (pred x) y)

plus' :: Integral i => i -> i -> i
plus' x y = if x >= 0
    then plus x y
    else negate (plus (negate x) (negate y))

times' :: Integral i => i -> i -> i
times' x y = if (x < 0 && y < 0) || (x >= 0 && y >= 0)
    then times (abs x) (abs y)
    else negate (times (abs x) (abs y))
```

Řeš. 2.1.11 Pro zadané číslo n funkce přičte $2n-1$ k výsledku rekurzivního volání pro vstup o jedna menší. Ten vrátí $2(n-1)-1$ a součet rekurzivního volání o 1 menší. To se bude dít tak dlouho, než vstup bude 0, pro nějž funkce vrátí 0. Tedy výsledkem bude součet

$$(2n-1)+(2(n-1)-1)+(2(n-2)-1)+\ldots+(2-1)+0.$$

To lze zapsat přesněji též jako

$$\sum_{i=1}^{n}(2i-1).$$

Předchozí výraz je perfektně správné řešení úlohy. Nicméně s použitím trochy matematiky lze řešení zapsat i elegantněji, aby bylo opravdu vidět, co zadaná funkce počítá.

Odečtení n jedniček lze vytknout za celý součet, a tedy výsledek lze zapsat i jako

$$\left(\sum_{i=1}^{n}2i\right)-n.$$

Násobení dvojkou lze též vytknout před součet, a tedy výsledek lze zapsat i jako

$$2\left(\sum_{i=1}^{n}i\right)-n.$$

Ze střední školy možná víte, že součet aritmetické řady $\sum_{i=1}^{n} i$ je $\frac{n\cdot(n+1)}{2}$. Takže výsledek lze zapsat též jako

$$2\frac{n\cdot(n+1)}{2}-n=n\cdot(n+1)-n=n\cdot n=n^2.$$

Pokud si chcete procvičit látku z Matematických základů informatiky, můžete si zkusit právě odvozenou rovnost

$$\sum_{i=1}^{n}(2i-1) = n^2$$

dokázat matematickou indukcí.

Řeš. 2.2.1 Stačí použít funkci `(:)`. Při otypování funkce musíme uvážit, že všechny prvky seznamu musí mít stejný typ, a to i nově vložená dvaačtyřicítka. Chtělo by se říci, že `42` je typu `Integer`, ale z předchozích příkladů již víme, že neotypovaný číselný literál (tj. číslo objevující se ve zdrojovém kódu) se může chovat jako libovolné číslo z typové třídy `Num`. Proto:

```
add42 :: Num n => [n] -> [n]
add42 xs = 42 : xs
```

Řeš. 2.2.2 Případ prázdného seznamu nemusíme řešit. Pro jednoprvkový seznam vrátíme rovnou jeho poslední prvek:

```
getLast [x] = x
```

Všechny zbývající případy seznamů mají alespoň dva prvky. Jednoduchá úvaha vede k tomu, že poslední prvek seznamu, který má alespoň dva prvky, je stejný jako poslední prvek téhož seznamu, ale bez prvního prvku. Tedy ze vstupního seznamu odstraníme první prvek a na zbytek aplikujeme rekurzivně funkci `getLast`:

```
getLast (x : xs) = getLast xs
```

Řeš. 2.2.3 Funkci definujeme obdobně jako funkci `getLast`. Začneme jednoprvkovým seznamem, kdy výsledkem je prázdný seznam:

```
stripLast [x] = []
```

Všechny zbývající případy seznamů mají dva nebo více prvků. V takovém případě bude první prvek zadaného seznamu určitě ve výsledném seznamu a zbytek lze vypočítat rekurzivně:

```
stripLast (x : xs) = x : stripLast xs
```

Srovnejte s definicí funkce `getLast`.

Řeš. 2.2.4 Délka prázdného seznamu je 0. Délka alespoň jednoprvkového seznamu je o 1 větší, než délka vstupního seznamu bez prvního prvku.

```
len :: [a] -> Integer
len []       = 0
len (_ : xs) = 1 + len xs
```

Výpočet této funkce probíhá například takto:

```
len (1 : (2 : [])) ⇝ 1 + len (2 : []) ⇝ 1 + (1 + len [])
    ⇝ 1 + (1 + 0) ⇝* 2
```

Řeš. 2.2.5

```
nth :: Int -> [a] -> a
nth 0 (x : _)  = x
nth n (_ : xs) = nth (n - 1) xs
```

Řeš. 2.2.6 Prázdný seznam neobsahuje nic. Neprázdný seznam obsahuje zadaný prvek právě tehdy, když je onen prvek prvním prvkem zadaného seznamu, nebo ho obsahuje zbytek zadaného seznamu.

```
contains :: Eq a => [a] -> a -> Bool
contains []       _ = False
contains (x : xs) e = e == x || xs `contains` e
```

Řeš. 2.2.7 Myšlenka je podobná jako u funkce `contains`, jen je potřeba si počítat, kolikrát zadaný prvek ještě chceme vidět. Pokud se dostaneme na 0, číslo už jsme viděli dostatečněkrát, a vrátíme tedy `True`.

```
containsNtimes :: Eq a => Integer -> a -> [a] -> Bool
containsNtimes 0 _ _        = True
containsNtimes _ _ []       = False
containsNtimes n x (y : ys) = if x == y
    then containsNtimes (n - 1) x ys
    else containsNtimes n x ys
```

Řeš. 2.2.8 Funkce `getPoints` je podobná funkci `contains`, ale místo logické hodnoty budeme vracet příslušnou hodnotu.

```
getPoints :: String -> [(String, Integer)] -> Integer
getPoints _ [] = 0
getPoints wanted ((name, points) : xs) = if wanted == name
    then points
    else getPoints wanted xs
```

U funkce `getBest` se hodí definovat si pomocnou funkci, která jako další argument dostane i jméno a počet bodů aktuálně nejlepšího studenta. Tohoto aktuálně nejlepšího studenta bude v průběhu vypočtu měnit a na konci seznamu ho vrátí.

```
getBest :: [(String, Integer)] -> String
getBest (x : xs) = fst (getBestWithDefault x xs)
    where
        getBestWithDefault current [] = current
        getBestWithDefault (curN, curP) ((newN, newP) : xs) =
            if newP > curP
            then getBestWithDefault (newN, newP) xs
            else getBestWithDefault (curN, curP) xs
```

Řeš. 2.2.9 Nejjednodušší je funkci definovat podle vzoru na prvním argumentu. Pokud je první seznam prázdný, výsledkem je přímo druhý seznam. Pokud je první seznam neprázdný, výsledný seznam obsahuje první prvek prvního seznamu a pak zřetězení zbytku prvního seznamu s druhým seznamem.

```
append :: [a] -> [a] -> [a]
append [] ys = ys
append (x : xs) ys = x : append xs ys
```

Řeš. 2.2.10 Zde se hodí vzor pro seznamy délky alespoň dva, protože potřebujeme pojmenovat první dva prvky vstupního seznamu. Poté stačí udělat z nich dvojici a zbytek seznamu vyřešit rekurzivně.

```
pairs :: [a] -> [(a, a)]
pairs (x : y : s) = (x, y) : pairs s
pairs  _          = []
```

Řeš. 2.2.11 a) Součet prázdného seznamu je 0. Součet neprázdného seznamu je součet prvního prvku a součtu zbytku seznamu.

```
listSum :: Num n => [n] -> n
listSum []       = 0
listSum (x : xs) = x + listSum xs
```

b) Existují nejméně dva přístupy k řešení, pokud nechceme explicitně pracovat s délkou seznamu. Jeden z nich je, že využijeme vzoru `(x : y : zs)`, který bere ze seznamu *po dvou* prvcích, tím pádem víme, že pokud tak skončíme na jednom prvku, seznam musel obsahovat lichý počet prvků:

```
oddLength :: [a] -> Bool
oddLength [] = False
oddLength [_] = True
oddLength (_ : _ : zs) = oddLength zs
```

Druhý přístup k řešení je, že odpověď postupně *vyskládáme* z prázdného seznamu, protože víme, že ten obsahuje sudý počet prvků. Každým dalším prvkem odpověď změníme na opačnou, s posledním prvkem získáme odpověď pro celý seznam:

```
oddLength' :: [a] -> Bool
oddLength' [] = False
oddLength' (_ : xs) = not (oddLength xs)
```

c) Opět přímočará rekurzivní definice funkce podle vzorů.

```
add1 :: Num n => [n] -> [n]
add1 [] = []
add1 (x : xs) = (x + 1) : add1 xs
```

d) Opět přímočará rekurzivní definice funkce podle vzorů.

```
multiplyN :: Num n => n -> [n] -> [n]
multiplyN _ [] = []
multiplyN n (x : xs) = (n * x) : multiplyN n xs
```

e) Opět přímočará rekurzivní definice funkce podle vzorů.

```
deleteEven :: Integral i => [i] -> [i]
deleteEven [] = []
deleteEven (x : xs) = if even x
    then deleteEven xs
    else x : deleteEven xs
```

f) Opět přímočará rekurzivní definice funkce podle vzorů.

```
deleteElem :: Eq a => a -> [a] -> [a]
deleteElem _ [] = []
deleteElem n (x : xs) = if x == n
    then deleteElem n xs
    else x : deleteElem n xs
```

g) Opět přímočará rekurzivní definice funkce podle vzorů.

```
largestNumber :: [Integer] -> Integer
largestNumber [x] = x
largestNumber (x : xs) = x `max` largestNumber xs
```

h) Stačí si uvědomit, jaké všechny případy mohou nastat. Jediný zajímavý případ je, když oba vstupní seznamy jsou neprázdné. V takovém případě je potřeba porovnat první prvky obou seznamů a také rekurzivně porovnat zbytky obou seznamů.

```
listsEqual :: Eq a => [a] -> [a] -> Bool
listsEqual [] [] = True
listsEqual [] _ = False
listsEqual _ [] = False
listsEqual (x : xs) (y : ys) = x == y && listsEqual xs ys
```

i) Tentokrát jen trochu komplikovanější rekurzivní definice funkce podle vzorů.

```
multiplyEven :: [Integer] -> [Integer]
multiplyEven [] = []
multiplyEven (x : xs) = if even x
    then (2 * x) : multiplyEven xs
    else multiplyEven xs
```

j) Tentokrát jen trochu komplikovanější rekurzivní definice funkce podle vzorů.

```
sqroots :: [Double] -> [Double]
sqroots [] = []
sqroots (x : xs) = if x > 0
    then sqrt x : sqroots xs
    else sqroots xs
```

Řeš. 2.2.12 Řešení se podobá funkci `nth`, při dosažení nuly se ale pokračuje dál s restartovaným čítačem. Povšimněte si, že v následující implementaci používá lokálně definovaná funkce `everyNthOffset` hodnotu `n`, která je argumentem vnější funkce.

```
everyNth :: Integer -> [a] -> [a]
everyNth n xs = everyNthOffset xs 0
    where
        everyNthOffset [] _ = []
        everyNthOffset (x : xs) 0 = x : everyNthOffset xs (n -
        ↪ 1)
        everyNthOffset (x : xs) m = everyNthOffset xs (m - 1)
```

Řeš. 2.2.13 Myšlenka řešení je procházet řetězec a držet si čítač právě otevřených a dosud neuzavřených závorek. Ten na konci musí být nulový a zároveň nikdy během průchodu nesmí klesnout pod nulu.

```
brackets :: String -> Bool
brackets s = bracketsWithDiff s 0
    where
        bracketsWithDiff [] k = k == 0
        bracketsWithDiff ('(' : xs) k = bracketsWithDiff xs (k +
        ↪ 1)
        bracketsWithDiff (')' : xs) k = k > 0 &&
                                        bracketsWithDiff xs (k -
                                        ↪ 1)
```

Řeš. 2.2.14 Funkci, která rozhodne, jestli je řetězec palindromem, zadefinujeme jednoduše pomocí funkce `reverse` a porovnání.

```
palindrome :: String -> Bool
palindrome str = str == reverse str
```

Po krátkém zamyšlení zjistíme, že na doplnění slova na palindrom nám stačí najít nejdelší příponu slova, která tvoří palindrom. Vynechané znaky ze začátku pak doplníme i na konec

řetězce v obráceném pořadí.

```
palindromize :: String -> String
palindromize s = if palindrome s
    then s
    else [head s] ++ palindromize (tail s) ++ [head s]
```

Poznámka: Vzhledem k častému využívání sekvenčního spojování seznamů (`++`) nemá tato funkce optimální časovou složitost. Zkuste se zamyslet, jak by se dala napsat efektivnější funkce.

Řeš. 2.2.15

```
getMiddle :: [a] -> a
getMiddle xs = tortoiseRabbit xs xs
    where
        tortoiseRabbit (t : _) [_] = t
        tortoiseRabbit (t : _) [_, _] = t
        tortoiseRabbit (_ : ts) (_ : _ : rs) = tortoiseRabbit ts
        ↪ rs
```